Abstraction Relationships for Real-Time Specifications

Monica Brockmeyer
Computer Science Department
Wayne State University
Detroit, MI 48202
mab@cs.wayne.edu

Abstract

This paper introduces the use of abstraction relationships for timed automata. Abstraction relations make it possible to determine when one specification implements *another, i.e. when they have the same set of computations. The approach taken here permits the hiding of* internal *events and takes into account the timed behavior of the specification. A new representation of the semantics of a specification is introduced. This representation,* min-max *automata is more compact than other types of finite state automata typically used to represent real-time systems, and can be used to define a variety of abstraction relationships.*

1 Introduction

This paper describes the use of *min-max* automata to specify the behavior of real-time systems compactly. Originally developed [2] as an alternative representation of timed behavior for the Modechart language[12], in order to support the evaluation of abstraction relationships between Modechart specifications, min-max automata are a general construct for representing the behavior of timed systems. Min-max automata are a more general form of automata than the computation graphs originally developed for Modechart[27], but are more compact than other types of automata which represent the passage of each unit of time as a distinct edge. Thus, min-max automata are more suitable for model-checking and automated evaluation of abstraction relationships between automata.

Abstraction and *refinement* relationships permit the evaluation of whether one specification can replace another. When can one specification replace another? What does it mean for two specifications to have the same behavior or for one specification to have more general behavior? Abstraction permits the substitution of module with a simpler implementation for one that is more complex. In abstraction, modules can be simplified by hiding internal details or by simplifying timing constraints.

There are several important uses for abstraction relations. This work was primarily motivated by the desire to ameliorate the state-space explosion problem which arises in mechanical model-checking. If one specification is an abstraction of another (i.e. it has more general behavior), then all behav-

iors of original are behaviors of the abstraction. Therefore, it may be advantageous to mechanically verify the abstraction rather than the original specification, should it have a more compact representation. Frequently, abstractions are created in an *ad hoc* manner in order to perform model checking. This research provides a formal basis for creating and using abstractions for real-time specifications.

Two other scenarios for using abstraction relations merit discussion. First, abstractions may be applied as part of a "top-down" development procedure. First, a very general specification of a real-time system may be defined. Then, a series of refinements may add increasing detail, resulting in specifications which are more operational. If this sequence of refinements is designed while maintaining an abstraction relation at each step, then properties which have been verified at

In particular, for real-time systems, the process of refinement might include the specification of tighter and tighter timing bounds as assumptions about the environment of a system are refined. the previous step will hold for each refinement step.

The last scenario involves showing an abstraction relationship between two specifications where one represents an implementation and the other represents the properties which must hold. In this case, instead of performing model-checking, one shows that a property, described as a specification, holds for the implementation.

Because of the timed behavior of Modechart specification, it is not possible to use the standard notion of program equivalence used to relate untimed concurrent programs [23]. The usual approach relies on the representation of the system as a *labeled transition system*. The original behavior representation of a Modechart specification [27], a computation graph, is a type of labeled transition system which captures the untimed behavior of a Modechart specification. Timing information is described in associated separation graphs. As a consequence it is not possible to define abstraction relationships directly for computation graphs.

The approach taken here is to represent all timing constraints explicitly in the labeled transition system. Then, the simulation relationships described in the literature can be directly applied. A new type of labeled transition system, *min-max timed automata*, are introduced. Each edge in the automata represents either the passage of time or a discrete system event which takes no time. Min-max automata represent elapsed time with time-passage edges which specify the minimum and maximum amount of time which can elapse between two discrete events.

The rest of this paper is organized as follows: Section 2 introduces both discrete-timed automata and min-max automata. Section 3 describes the extensions to the usual definitions for a *move* in an automata necessary to define abstraction and simulation relationships. Section 4 defines bisimulation and trace inclusion relationships for min-max automata. Conclusions and future work are found in Section 5.

2 Definition of Min-Max Automata

This research is motivated by two goals. First is the ability to mechanically evaluate abstraction relationships between automata representing timed systems. Second is achieving a compact representation of timed systems. These goals are achieved by the use of

min-max automata in which each time passage edge denotes a *range* of possible times elapsed. This results in a more compact representation than other approaches because multiple paths can be collapsed into one.

However, because each path in the min-max automata can potentially represent more than one timed execution of a system, the usual notions of bisimulation and abstraction relations cannot be directly applied.

Definition 2.1. A min-max automata, A is defined as the tuple

$< states(A), initial(A), actions(A), next(A) >$ where

- The initial states, $initial(A) \subseteq states(A)$,
- The actions of A, $actions(A)$, is the union of the the sets $external(A)$ and $\{\tau\}$ and $times(A) = \{(min, max) : min, max \in \{\mathbf{Z}^+ \cup \infty\}$ and $min \leq max\}$ where τ is called the internal action and $times(A)$ are time-passage actions, and
- The next-state relation, $next(A)$ is a subset of $states(A) \times actions \times states(A)$.

Min-max automata, like discrete timed automata, are examples of Lynch's [22] untimed automata. And like discrete-timed automata, the time-passage actions can be used to assign occurrence times to external events in a trace to form a computation.

τ is distinguished as the *internal* action of A. It is considered to be invisible outside of A. If σ is a sequence of actions in $actions(A)$, then $\hat{\sigma}$ is the same sequence with all τ actions removed, and $\bar{\sigma}$ is the sequence with the time actions (elements of $times(A)$) removed.

If $(s, a, s') \subset next(A)$, then the notation $s \xrightarrow{a} s'$ may be used to indicate this. If there is a sequence, σ for which there are states $s_0, s_1, s_2, \ldots s_n$, such that for all i, $s_i \xrightarrow{\sigma_i} s_{i+1}$, σ is called a finite execution fragment of A, and one can write $s_0 \xrightarrow{\sigma} s_n$. For an infinite sequence, the notation $s_0 \xrightarrow{\sigma}$ is used. A *move* of A, indicated by $s \stackrel{\gamma}{\Longrightarrow} s'$, occurs if $s \xrightarrow{\sigma} s'$ and $\gamma = \hat{\sigma}$. Thus, a move ignores internal actions.

If s_0 is an initial state of A, then σ is an execution of A. The sets, $execs^*(A), execs^\omega$, and $execs(A)$ indicate the sets of finite, infinite, and executions of A. If the time passage actions are removed, $(\bar{\sigma})$, the resulting sets are the untimed finite, untimed infinite, and untimed executions of A, denoted $execs^*_U(A), execs^\omega_U$, and $execs_U(A)$. If the internal action is removed from an execution of A, $\gamma = \hat{\sigma}$, the resulting sequence is called a trace of A. The sets of traces of A are $traces^*(A), traces^\omega(A)$, and $traces(A)$ for the finite, infinite, and all traces of A. The corresponding untimed traces, $traces^*_U(A), traces^\omega_U(A)$, and $traces_U(A)$ are also defined, for the corresponding $\bar{\gamma}$.

The actions in the set $external(A)$ represent the discrete, externally visible actions of the system. In the context of Modechart, these could represent mode entry, mode exit, and mode transition events which are visible on the interface of a Modechart module. The symbol τ is used to represent internal events which can not be observed externally. Both $external(A)$ and τ events occur instantaneously. The set $external(A) \cup \{\tau\}$ is called $discrete(A)$.

The time passage actions represent the passage of an amount of time between the values of min and max. When they occur in an execution, they represent time elapsing between the instantaneous external and τ events. The values of a time-passage edge, e, are indicated by $min(e)$ and $max(e)$. A *timed event sequence* is a se-

quence $\delta = d_0, d_1, d_2, \ldots$ with $d_i = (a_i, t_i) \in discrete(A) \times \mathbf{Z}^+$ and t_i increasing. If the timed event sequence corresponds to some execution σ of A such that for every $d_i \in \delta$, if $a_i = \sigma_k$ then $t_i \geq \sum_{\substack{0 \leq j < k \\ \sigma_j \in times(a)}} min(\sigma_j)$ and $t_i \leq \sum_{\substack{0 \leq j < k \\ \sigma_j \in times(a)}} max(\sigma_j)$ if $max_j \neq \infty$ for all j. If $max_j = \infty$ for any j, then only the lower bound restriction holds. then δ is called a *timed execution* of A. It can be observed that δ assigns times to the discrete events in σ in a way that is consistent with the time passage events in σ. If the timed event sequence corresponds to a trace of A it is called a *timed computation*. $comps(A)$ indicates the set of timed computations of A.

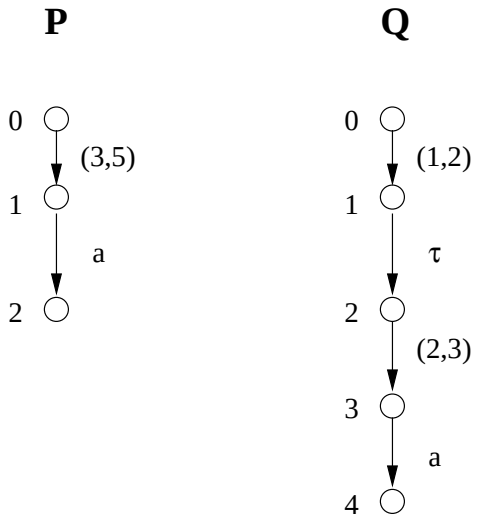

Figure 1: Complications in matching time passage edges in a min-max automata

3 Issues in Defining Abstraction Relations for Min-Max Automata

Direct application of the definitions for abstraction relations described in the literature is problematic, since each path through a min-max automata represents more than one (timed) computation. As a consequence, soundness and completeness results which hold for the ordinary definitions of abstraction relationships (e.g. bisimulation) will hold for traces of min-max automata, but not necessarily for computations.

Moreover, time-passage edges have some properties which cause unexpected results when the ordinary abstraction relations are applied directly using the usual definition of a move. The definition of a move is relaxed, leading to more powerful abstraction relations.

Example 3.1. Consider the min-max automata P and Q, depicted in Figure 1.

The ordinary definition of a move will not permit the sequence $0 \longrightarrow_P 1, 1 \longrightarrow_P 2$, to be matched to $0 \Longrightarrow 4$ in Q, for any of the abstraction relations described. Yet, the two systems describe the same set of timed computations and have very similar structure.

It should be possible to extend the definition of a move to permit a single time-passage edge to be matched with an appropriate sequence of time-passage edges in the abstraction such that a time passage edge on (m, n) could be matched by a sequence of time passage edges whose minimums sum to m and whose maximums sum to n. However, the definitions of abstraction relationships described in the literature match a single edge to a move. That is, if a min-max automata has an edge with action $(1, 2)$ followed by $(2, 3)$, while it can be said that the automata moves on $(3, 5)$, what move should each of $(1, 2)$ and $(3, 5)$ be matched to in the abstraction automata?

Other approaches (discrete-timed automata[3] and [22] for example) ad-

dress this problem by filling in all the possible time passage edges. In this case, if there were an edge $(3,5)$ in a min-max automata between points s and s', then there would have to be every possible sequence of edges between s and s' such that the sum of the minimum times was 3 and the sum of the maximum times was 5. However, this defeats the purpose of min-max automata which is to provide a finite and more compact representation of a system, by using min-max time passage edges.

Instead, the problem is addressed by defining a canonical representation for a system. The canonical representation combines all sequences of time-passage edges and replaces them with new edges corresponding to a move. In the example, the sequence $(1,2),(2,3)$ would be replaced by a single time-passage edge $(3,5)$. The abstraction relations are then defined on the canonical representation. A *canonical* representation of a min-max automata, A, denoted $can(A)$, is defined by computing the closure of a min-max automata with regard to the time-passage edges and deleting all but the maximal length edges.

A consequence of computing the canonical representation of a min-max automata is that some points are left unreachable. Since the canonical representation represents the same set of timed computations as the original min-max automata, this is of no consequence. However, the definitions of the simulation relationships must be adjusted to take this into account. The unreachable points are not required to be included in the simulation relations.

Definition 3.1. A point s in a min-max automata is *reachable* if there is a sequence σ such that $s_0 \stackrel{\sigma}{\longrightarrow} s$, where s_0 is an initial point of the automata. The set of reachable points of an automata A is denoted $reachable(A)$.

The abstraction relations will be defined almost identically as in the literature. However, only reachable points will be included and the canonical representation of the min-max automata will be used. This will address the anomaly from Example 3.1.

A second issue is described in Example 3.2.

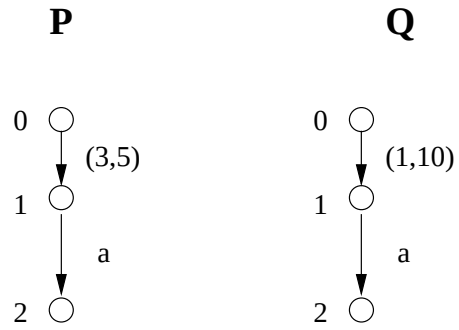

Figure 2: Rationale for a relaxed-time move in a min-max automata

Example 3.2. Consider min-max automata P and Q, depicted in Figure 2.

Then $comps(P) \subset comps(Q)$, but there is no abstraction relationship between P and Q. If the individual computations were represented on separate paths as they are for discrete-timed automata, then an abstraction relation would exist.

This problem is avoided by extending the definition of a move, to permit time-passage edges to be matched to time-passage edges which are inclusive of the times represented by the original edge. That is, a time-passage edge (m,n) will be matched to a time-passage edge (m',n') if $m' \leq m$ and $n \leq n'$.

First, a *time-relaxed step*, relaxes the timing requirements of a time-passage edge.

Definition 3.2. If $s \stackrel{(m,n)}{\longrightarrow} s'$, and $m' \leq m$ and $n \leq n'$, then $s \stackrel{(m',n')}{\mapsto} s'$

Next, the definition of a move is expanded to accommodate time-relaxed steps.

Definition 3.3. A *time-relaxed move* of A, indicated by $s_0 \overset{\gamma}{\rightsquigarrow} s_n$, occurs if $\gamma = \hat{\sigma}$ where σ is a sequence of states, $s_0, s_1, s_2, \ldots s_n$, such that for all i, $s_i \xrightarrow{\sigma_i} s_{i+1}$ or $s_i \overset{\sigma_i}{\mapsto} s_{i+1}$.

By substituting time-relaxed moves for ordinary moves in the definitions of the abstraction relations, the anomaly described in Example 3.2 is avoided. It is now possible to define abstraction relations for min-max automata.

4 Abstraction Relations for Min-Max Automata

This paper now considers the issues of when one specification is an abstraction (or implements) another specification. Trace inclusion or trace equivalence has been widely used to describe when one system implements another [21, 22]. The terms simulation [21], homomorphism [18], and refinement mapping [1] have all been used to reduce the problem of showing trace inclusion to proving something about transitions in some kind of automata. Thus, only a local property needs to be demonstrated. All of these techniques relate systems in terms of the timed behavior of visible events. In each case, the behavior of internal events is hidden. This section describes several such relationships in the context of discrete-timed automata.

4.1 Bisimulation and Forward Simulation

One common technique for showing that two systems are observationally equivalent is called *bisimulation* [25]. Bisimulation involves finding a relation on the states of two systems such that two states being bisimilar means that each state has an edge to a state so that the resulting states are bisimilar. This approach can be relaxed (called *weak* bisimulation) so that an edge in each system is matched by a *move* (including internal events) so that the resulting states are bisimilar. Bisimulation is a rather conservative notion of system equivalence, as it is sound but not complete, but it is widely used especially in process algebras [24].

In order to hide internal events, a sequence of steps, or a *move* is more relevant to the question of whether two automata similarly. A move, as defined above, is a subpath between two points where no intervening events are externally visible. A *weak bisimulation* [25] relaxes the requirement that the two systems proceed in lockstep. Rather, it is only necessary that an edge between two points correspond to a move between two points.

Definition 4.1. For min-max automata, P and Q, $r \subset reachable(can(P)) \times reachable(can(Q))$, is a *weak bisimulation*, if

- for all $p \in initial(P)$, there is some q, such that $(p, q) \in r$ and $q \in initial(Q)$,
- for all $q \in initial(Q)$, there is some p, such that $(p, q) \in r$ and $p \in initial(P)$,
- if $\forall (p, q) \in r$:
  - if $p \xrightarrow{e} p'$ then $\exists q' : q \overset{e}{\rightsquigarrow} q'$ and $(p', q') \in r$, and
  - if $q \xrightarrow{e} q'$ then $\exists p' : p \overset{e}{\rightsquigarrow} p'$ and $(p', q') \in r$.

Informally, this states that two points are bisimilar if any edge from one of the points

can be matched by the other point making a move on the same event and reaching a point that is weakly bisimilar to the point reached from the first point. Since weak bisimulations are closed under union, it can be shown that there is a largest weak bisimulation, denoted $\approx$, for any pair of computation graphs for a given set of observable events.

The following theorem establishes the soundness of bisimulation.

Definition 4.2. The notation $comps(P) \equiv comps(Q)$ indicates $comps(P) \subseteq comps(Q)$ and $comps(Q) \subseteq comps(P)$.

Theorem 4.1. $P \approx Q \implies comps(P) \equiv comps(Q)$.

Proof. Similar to the proof for ordinary timed automata found in the literature [22]. The proof is in [2], which shows that the extensions to the definition of a move do not violate the conditions of the usual proof. □

Bisimulation is not complete. That is, there are systems which have the same set of timed traces, but which are not bisimilar. This is because bisimulation captures some aspects of system structure. Each point must be bisimilar to a point in the other system which permits actions which move to points which are bisimilar to those which can occur in the original specification. As a consequence, bisimulation distinguishes with regard to the state of the system as well as the sequence of actions or events.

4.2 Forward Simulations

If the definition of bisimulation is modified to apply in only one direction, the result is called a *forward simulation* [21]. Forward simulations are also related to simulations [28, 13], history measures [17], downward simulations [9, 11, 15], and possibilities mappings [20]. Because the restriction is in one direction, a forward simulation shows trace inclusion rather than trace equivalence.

In practice, this approach is desirable. Often a general purpose specification will be designed as well as an implementation or operational specification which has a narrower set of behaviors. It is not necessary for the implementation to have the full set of behaviors as the specifications. Alternatively, perhaps a simplification can be made to a specification which reduces the size of the computation graph, but which admits a larger set of behaviors. If the a trace inclusion relationship holds between the two systems, then it may be possible to model-check the simpler system and apply the results to the more complicated system.

Definition 4.3. For min-max automata, *forward* simulation from P to Q is a relation f over $reachable(can(P))$ and $reachable(can(Q))$ a *forward* simulation if:

- for all $p \in initial(P)$, there is some q, such that $(p, q) \in f$ and $q \in initial(Q)$,
- if $\forall (p, q) \in f$ and all $e \in actions(P)$, $p \xrightarrow{e} p'$ then $\exists q' : q \overset{e}{\rightsquigarrow} q'$ and $(p', q') \in f$.

Lynch [21] shows that forward simulations are a pre-order (i.e. they are reflexive and transitive). Soundness follows from the soundness of bisimulations.

4.3 Forward-Backward Simulations

Forward-Backward simulations were also described by Lynch and are similar to the invariants and ND-measures of [16, 17] as well

as subset simulations [14], and simple failure simulations [7]. They are less restrictive than forward simulations. Perhaps, most noteworthy is that they are complete for trace inclusion. However, since a single trace of a min-max automata can represent more than one timed computation, forward-backward simulations are not complete for timed computations.

Definition 4.4. For min-max automata, *forward-backward* simulation from P to Q is a relation fb over $reachable(can(A))$ and $\mathbf{N}(reachable(can(B)))$ [1] such that:

- for all $p \in initial(P)$, there is some set A, such that $(p, A) \in fb$ and $A \subseteq initial(Q)$,
- if $p \xrightarrow{e} p'$ and $(p, A) \in fb$, then there exists a set A' such that $(p', A') \in fb$ such that for every $q' \in A'$ there is some $q \in A$ such that $q \overset{e}{\rightsquigarrow} q'$.

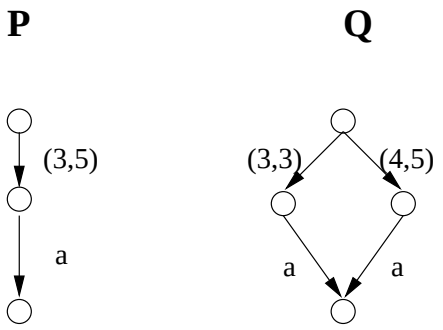

Figure 3: Completeness Problem for Min-Max Automata

Example 4.1. To understand why forward-backward simulations are not complete for min-max automata, consider min-max automata, P and Q, depicted in Figure 3.

[1] For a set X, $\mathbf{N}(X)$ indicates the set of non-empty subsets of X.

Then, $comps(P) \equiv comps(Q)$ but it is not the case that $P \leq_{FB} Q$, because there is no match for the time-passage edge, $(3, 5)$.

Therefore, further research is required to find an abstraction relation which is complete for computations of min-max automata.

4.4 Homomorphisms and Refinements

Homomorphisms [8, 18] and refinement mappings [1, 19, 21], are more restrictive than forward simulations, because they require a *function* from $states(P)$ to $states(Q)$ rather than a relation.

Definition 4.5. For min-max automata, P and Q, a function f between $reachable(can(P))$ and $reachable(can(Q))$, is a *refinement* if:

- for all $p \in initial(P)$, $f(p) \in initial(Q)$,
- if for all $e \in actions(P)$ $p \xrightarrow{e} p'$ then $f(p) \overset{e}{\rightsquigarrow} f(p')$

The proof of soundness for forward simulations, forward-backward simulations, and refinements is similar to that for bisimulations.

Another interesting type of relationship between two automata is failures inclusion or equivalence, developed by Hoare [4, 10]. An alternative characterization, given by Hennessy and de Nicola [6], is called testing equivalence in which equivalent automata pass or fail the same set of tests. Testing and failures relationships cannot be characterized by matching an edge in one automata with some kind of move in another automata and so are not discussed in this paper.

5 Conclusions and Future Work

This paper has introduced *min-max automata* which are a compact form of timed automata suitable for mechanical evaluation of simulation and abstraction relationships. Extensions to the definition of a move necessary to support simulation and abstraction relationships were defined and several types of equivalence and abstraction/simulation relationships were described in the context of min-max automata. Related research efforts extend these ideas by describing automatic generation of abstractions [2].

Future work involves integration of min-max automata into existing software tools to automatically generate min-max automata for Modechart specifications and to automatically check for the simulation and abstraction relationships defined in this paper. The Modechart Toolset [5, 26] provides a graphical interface for editing, consistency-checking, simulation, and verification of real-time specifications in the Modechart Language. This will permit evaluation of the techniques on real-world examples. Future work is also required to define an abstraction relationship which is complete for trace inclusion of min-max automata.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–281, 1991.

[2] M. Brockmeyer. *Monitoring, Testing, and Abstractions of Real-Time Specifications*. PhD thesis, The Department of Electrical Engineering and Computer Science, The University of Michigan, 1999.

[3] M. Brockmeyer. Using modechart modules for testing formal specifications. In *Proceedings of the High Assurance Systems Engineering Workshop*. IEEE, 1999.

[4] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of ACM*, pages 560–599, 1984.

[5] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. MT: A toolset for specifying and analyzing real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1993.

[6] R. de Nicola and M. C. Hennessy. Testing equivalences for processes. *Journal of Theoretical Computer Science*, pages 83–133, 1983.

[7] R. Gerth. Foundations of compositional program refinement. In *Proceedings REX Workshop on stepwise refinement in distributed systems: Models, Formalism, Correctness, Lecture Notes in Computer Science*, volume 430, pages 777–808, 1987.

[8] A. Ginzburg. *Algebraic Theory of Automata*. Academic Press, 1968.

[9] J. He. Process simulation and refinment. *Journal of Formal Aspects of Computing Science*, 1:229–241, 1989.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[11] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.

[12] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Software Engineering*, 20(10), 1994.

[13] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Upsaala University, 1987.

[14] B. Jonsson. Simulations between specifications of distributed systems. In *Proceedings Concur '91, Lecture Notes in Computer Science*, volume 527, pages 347–360. Springer-Verlag, 1991.

[15] M. B. Josephs. A state-based approach to distributed processing. *Distributed Computing*, 3:9–18, 1988.

[16] N. Klarlund and F. Schneider. Verifying safety properties using infinite state automata. Technical Report 89-1039, Department of Computer Science, Cornell University, 1987.

[17] N. Klarlund and F. Schneider. Proving non-deterministically specified safety properties using progress measures. *Information and Computation*, 171(1):151–170, November 1993.

[18] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-theoretic Approach.* Princton University Press, 1994.

[19] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages*, 5:190–222, 1983.

[20] N. Lynch. Multivalued possibilities mappings. In *Proceedings REX Workshop on stepwise refinement in distributed systems: Models, Formalism, Correctness, Lecture Notes in Computer Science*, volume 430, pages 519–543, 1987.

[21] N. Lynch and F. Vaandrager. Forward and backward simulations – part i: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[22] N. Lynch and F. Vaandrager. Forward and backward simulations – part ii: Timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

[23] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[24] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[25] D. Park. Concurrency and automata on infinite sequences. *Lecture Notes in Computer Science*, 104, 1980.

[26] A. Rose, M. Perez, and P. Clements. Modechart toolset user's guide. Technical Report NRL/MRL/5540-94-7427, Center for Computer High Assurance Systems, Naval Research Laboratory, Washington, D.C., February 1994.

[27] D. Stuart. Implementing a verifier for real-time systems. In *Real-Time Systems Symposium*, pages 62–71, Orlando, FL, December 1990.

[28] R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, Free University, The Netherlands, 1990.